به نام خدا

مولد تحلیل گر لغوی و تجزیه گر نحوی

Compiler Generators

Lex AND Yacc ©

استاد راهنما:

دکتر سعید پارسا

دانشگاه علم و صنعت ایران

پاییز 83

چکیده: امروزه استفاده از مولدهای کامپایلر رواج بسیاری یافته است.به طوری که کمتر کسی برای تولید یک محصول تجاری یا علمی خودش اقدام به نوشتن برنامه پارسر و تحلیل گر لغوی می کند. در عوض با استفاده از مولدهای کامپایلر به سرعت و با صرف کمترین هزینه می توان یک کامپایلر با امکانات کامل داشت. در این مقاله برآن هستیم تا شما را با دو محصول رایج تجاری در این زمینه آشنا کنیم.Lex و Yacc امروزه به شکل های گوناگون و در نسخه های متعدد و با زبان هایی نظیر C،++C و Java در بازار نرم افزار یافت می شود و نسبت به سایر مولد های کامپایلر از محبوبیت و گستردگی نسبی برخوردار است.

کلمات کلیدی: Lex ، Yacc ، تحلیل گر لغوی، تحلیل گر نحوی، پارسر، UNIX، compiler compiler،Error Recovery

معرفی Lex , Yacc

یک قول معروف می گوید برای ساختن یک اتوموبیل جدید، چرخ را دوباره اختراع نکنید.
Parser Generator ها یا همان مولدهای تجزیه گر نحوی، برنامه هایی هستند که به ما کمک می کنند بدون نیاز به کدنویسی پیچیده یک کامپایلر برای هر زبانی که خودمان قواعدش را تعیین می کنیم، درست کنیم. در واقع مولد تجزیه گر نحوی ساختن کامپایلر را تسهیل می کند. از بین این برنامه ها ما در اینجا Yacc را که به طور گسترده ای در دنیا کاربرد دارد، بررسی می کنیم. Yacc مخفف عبارت "**Y**et **A**nother **C**ompiler **C**ompiler"می باشد. Compiler Compiler یعنی برنامه ای که یک کامپایلررا کامپایل می کند. یا به عبارت دیگر منظور همان مولد تجزیه گر نحوی است.Yacc یا Bison به عنوان یکی از Package های همراه Linux و Unix است و برای تولید کامپایلر های زیادی مورد استفاده قرار گرفته است.

Yacc مولد تجزیه گر نحوی

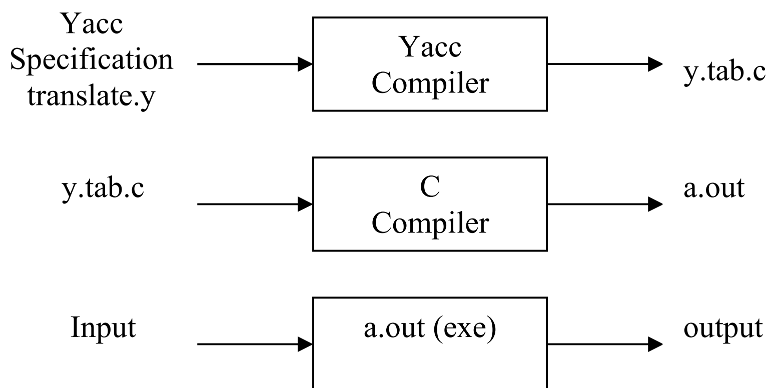
یک کامپایلر می تواند به روشی که در شکل زیر توضیح داده شده است، توسط Yacc تولید شود. در ابتدا یک فایل به نام translate.y که حاوی قواعد نحوی زبان مورد نظر شما است، با استفاده از دستور زیر در UNIX به یک فایل با پسوند C که حاوی دستوراتی به زبان C است، تولید می شود.

Yacc translate.y

این دستور، فایل translate.y را به یک برنامه C به نام y.tab.c ترجمه نموده که این کار براساس روش تولید جدول LALR انجام می پذیرد. در واقع فایل y.tab.c ارائه ای از پارسر LALR و تعدادی

از توابع و روتین های تعریف شده توسط کاربر به زبان C است. اما این فایل توسط انسان قابل فهم و خواندن نیست. بعد از کامپایل و لینک با استفاده از کتابخانه ly (Lex and Yacc) که توسط دستور زیر در UNIX انجام می شود، ما یک فایل خروجی به نام a.out دریافت خواهیم کرد که حاوی توابع لازم برای تجزیه(Parse) و بر اساس ویژگی ها و قواعدی است که ما قبلا در فایل translate.y داده بودیم. حال کافی است فایل a.out را به یک کامپایلر زبان C بدهیم تا برای ما فایل اجرایی exe. را تولید کند.

حالا این برنامه exe هر ورودی را برای ما براساس قواعد زبان تحلیل نحوی می کند.



اجزای یک برنامه Yacc:

یک برنامه Yacc از 3 قسمت تشکیل شده است که هر قسمت از قسمت پایین تر توسط علامت %% جدا می شود. چیزی شبیه این:

```
declarations
%%
translation rules
%%
supporting C-routines
```

مثال: برای اینکه بیان کنیم چگونه می توان قواعد یک زبان را در قالب یک برنامه منبع یا سورس کد Yacc بیان کرد، لطفا به این مثال توجه فرمایید.فرض کنید می خواهیم یک برنامه ماشین حساب بسازیم که از ورودی عبارات پرانتز گذاری شده را دریافت کرده، آن را ارزیابی کرده و در خروجی پاسخ صحیح را چاپ کند. و باز فرض کنید که ما گرامر زیر را برای این برنامه در نظر می گیریم:

E → E+T | T

T → T*F | F

F → (E) | **digit**

بخش declaration: دو قسمت اختیاری از دستورات در این بخش قرار دارد: در قسمت اول ما اعلان های معمولی C نظیر دستورات #include را قرار می دهیم که این قسمت با }% شروع شده و با {% پایان می پذیرد. در قسمت دوم این بخش ما Token های زبان را معرفی می کنیم. Token هایی که در این بخش معرفی می شوند می توانند در بخش های دوم و سوم مورد استفاده قرار بگیرند. برای تعریف Token از عبارت token% استفاده می کنیم.

بخش Translation rules:

در فایل yacc بعد از اولین علامت %% قواعد گرامر زبان را قرار می دهیم. هر قانون می تواند به شکل کلی زیر باشد:

< left side > → < alt 1 > | < alt 2 > | … | < alt n>

که در فایل yacc به صورت زیر نوشته می شود:

```
< left side > : < alt 1 >    { semantic action 1 }
              | < alt 2 >    { semantic action 2 }
              …
              | < alt n >    { semantic action n }
```

درهر نسخه ای از برنامه Yacc یک کاراکتر که در داخل ' ' قرار دارد، به عنوان یک ترم پایانی در نظر گرفته می شود. در صورتیکه هر رشته ای از حروف و اعداد که به عنوان یک token در بالا تعریف نشده باشد، یک ترم میانی در نظر گرفته می شود. سر ترم گرامر یا همان نماد آغازگر گرامر، اولین سمت چپ در فایل Yacc ما خواهد بود. در واقع اولین قاعده همیشه بیانگر نماد آغازگر یا سرترم گرامر است.

اما semantic action می تواند یک توالی از کد C یا توابع شناخته شده در C یا هر تابعی باشد که header مربوط به آن تابع در بخش اول فایل Yacc تعریف شده باشد. این قطعه کد ها زمانی اجرا

می شوند که قاعده مربوطه بخواهد کاهش داده شود. در واقع به محض مواجهه با رویداد کاهش توسط قاعده مربوطه این کد نیز اجرا خواهد شد.

لطفا به قطعه کد زیر که محتویات فایل yacc برنامه ماشین حساب است، توجه فرمایید:

```
%{
#include <ctype.h>
%}
%token DIGIT

%%

line    : expr   '\n'         { printf("%d\n",$1); }
        ;
expr    : expr  '+'  term     { $$=$1+$3; }
        | term
        ;
term    : term  '*' factor    { $$=$1*$3; }
        | factor
        ;
factor :  '(' expr ')'        { $$=$2; }
        | DIGIT
        ;
%%
yylex()
{
        int c;
        c = getchar();
        if ( isdigit (c) )
                {
                        yyval=c - '0' ;
```

```
                return DIGIT;
            }
        return c;
}
```

در بخش اول دستور #include<ctype.h> به preprocessor زبان C می گوید که header مربوطه را به فایل اضافه کند تا بتواند از تابع isdigit استفاده کند. در همین بخش %token DIGIT این ترم را به عنوان یک Token در نظر بگیرد.

دربخش بعد ما قواعد گرامری را که در بالا برای ماشین حساب معرفی کردیم، می بینیم. بعد از هر قانون ; آمده است که اتمام آن قاعده را نشان می دهد. در semantic action ها، $$ ارجاع به مقداری دارد که در لحظه کاهش یا reduce متغییر سمت چپ دارا می باشد. همینطور $1 اشاره به مقدار اولین ترم بعد از علامت : یا | در قاعده مربوطه دارد.به همین ترتیب $3 به سومین و ....

به عنوان مثال در قاعده زیر

```
expr   : expr  '+'  term     { $$=$1+$3; }
       | term
```

semantic action مربوطه مقدار expr ($1) و term ($3) را به هنگام کاهش جمع زده و در متغیر سمت چپ ($$) قرار می دهد.

توجه دارید که در اولین گسترش قاعده فوق، expr سمبل اول و '+' سمبل دوم و term سمبل سوم می باشد.البته اگر توجه داشته باشید برای گسترش دوم همین قاعده ما از نوشتن {$$=$1;} خودداری کرده ایم. علت این است که این عمل پیش فرض yacc است و به طور خودکار این کار انجام می شود.بنابراین لزومی به نوشتن آن نیست.

به قاعده اول این فایل توجه نمایید:

```
line    : expr   '\n'        { printf("%d\n",$1); }
        ;
```

ما در این قاعده گفته ایم که ورودی برای ماشین حساب ما یک عبارت محاسباتی است که بعد از آن '\n' آمده باشد. و البته در این گرامر line نماد اغازگر یا سرترم گرامر ماست.اما semantic action مربوطه چه کاری انجام می دهد؟ در واقع اگر توجه نمایید این خط برای ما حاصل عبارت ارزیابی شده را چاپ می کند. به محض اینکه عمل تجزیه با موفقیت انجام شود یا به عبارت دیگر به سر ترم گرامر برسیم این حاصل در خروجی چاپ خواهد شد. در این عبارت $1 مقدار expr در لحظه کاهش به سرترم گرامر است.

اما در بخش سوم همین مثال شما تابع تحلیلگر لغوی یا همان Lexical Analyser را مشاهده می نمایید. یک تابع تحلیلگر لغوی با نام ()yylex باید در هر فایل yacc موجود باشد. سایر روال ها مانند error recovery و ... نیز می توانند به ضرورت اضافه شوند ولی آنچه که الزامی است ، این تابع تحلیلگر لغوی است. تابع تحلیلگر لغوی جفت Token و مقدار مربوطه آن را تولید می کند. به عنوان مثال اگر عددی را مشاهده کند، ضمن برگرداندن DIGIT مقدار آن را در متغیر yyval برمی گرداند. این متغیر نیز توسط خود Yacc تعریف شده است. در تابع تحلیلگر لغوی این مثال، اگر یک عدد کشف شد، DIGIT به عنوان Token برگردانده می شود. در غیر این صورت، خود کاراکتر به عنوان Token برگردانده می شود.

اما چنانکه مشاهده می نمایید، نوشتن دستی تابع تحلیلگر لغوی در مثال های پیچیده تر ممکن است مشکل باشد. لذا در کنار برنامه Yacc برنامه دیگری به نام Lex نیز ساخته شده است تا به وسیله آن بتوان تابع تحلیلگر لغوی ساخت. بدیهی است که تابع ساخته شده توسط Lex می تواند به عنوان تحلیلگر لغوی yacc مورد استفاده قرار گیرد. کتابخانه ای در Lex به نام ll وجود دارد که درایوری به نام ()yylex فراهم می کند.این همان اسمی است که Yacc برای تحلیلگر لغوی خود به آن نیاز دارد.اگر Lex به عنوان مولد تحلیلگر لغوی مورد استفاده قرار بگیرد، تابع ()yylex در قسمت سوم فایل Yacc را با این عبارت جایگزین می کنیم:

#include “lex.yy.c”

و هر بار عمل Lex یک ترم پایانی شناخته شده در Yacc را بر می گرداند. با استفاده از عبارت ”#include “lex.yy.c ، برنامه ()yylex به نام های Yacc برای Token ها دسترسی دارد.زیرا خروجی Lex به عنوان بخشی از خروجی Yacc یعنی y.tab.c کامپایل می شود.

در محیط UNIX فرض کنید فایلی داریم به نام first.l که حاوی مشخصات تحلیلگر لغوی است و فایلی داریم به نام second.y که حاوی مشخصات تحلیلگر نحوی است.برای بدست آوردن پارسر مورد دلخواه خود، می توانیم دستورات زیر را وارد کنیم:

lex first.l

yacc second.y

cc y.tab.c –ly -ll

در Lex هم مانند Yacc فایل ما 3 قسمت دارد. در بخش اول Symbol های زبان معرفی می شوند.به عنوان مثال اگر بخواهیم به Lex بگوییم همه اعداد بین 0 تا 9 را به عنوان number برگردان، از دستور زیر استفاده می کنیم:

number   [0-9]+\.? | [0-9]*\.[0-9]+

اما برای اینکه مفهوم این عبارات بهتر درک شود، به بخش زیر که معرفی متاکاراکتر های مورد استفاده در Lex است، توجه فرمایید.

متاکاراکتر ها در Lex:

در Lex عبارات با قاعده یا در واقع همان الگوها توسط متاکاراکترها تعریف می شوند. در جدول زیر این متاکاراکتر ها به همراه مفهوم یا ساختار الگوی آنها آمده است:

Metacharacter	Matches
.	هر کاراکتری به غیر از '\n'
\n	newline
*****	صفر یا بیشتر از یک الگو
+	یک یا بیشتر از یک الگو
?	صفر یا بیشتر از عبارت قبل از آن
^	شروع خط
$	انتهای خط
a|b	a یا b
(ab)+	یکی یا بیشتر ab
[]	character class

Expression	Matches
abc	**abc**
abc*	**ab abc abcc abccc ...**
abc+	**abc, abcc, abccc, abcccc, ...**
a(bc)+	**abc, abcbc, abcbcbc, ...**
a(bc)?	**a, abc**

[abc]	one of: **a, b, c**
[a-z]	any letter, a through z
[a\-z]	one of: **a, -, z**
[-az]	one of: **- a z**
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: **a, b**
[a^b]	**a, ^, b**
[a|b]	**a, |, b**
a|b	**a, b**

در بخش دوم از فایل Lex الگوها و رفتاری که تحلیلگر باید در هنگام کشف آن الگو انجام دهد، را معرفی می کنیم.

در بخش سوم نیز می توانیم توابعی قرار دهیم که عمده ترین آنها تابع main() است که در هر برنامه C قرار دارد. همینطور تابع دیگری داریم به نام wrap که در انتهای کار تحلیلگر لغوی فراخوانی می شود. در مثال ساده زیر می توانید بخش های یک تحلیلگر لغوی را مشاهده کنید:

```
%{
   int yylineno;
%}
%%
^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
   yyin = fopen(argv[1], "r");
   yylex();
   fclose(yyin);
}
```

در این مثال در بخش اول متغیری به نام yylineno تعریف شده است.این متغیر شماره هر سطر از ورودی را نگهداری می کند. در بخش دوم الگویی معرفی شده است که به تحلیلگر لغوی می گوید از ابتدای سطر اگر هر تعداد کاراکتری به غیر از '\n' و در انتهای همه '\n' را دیدی، شماره سطر و

محتوای آن سطر را در خروجی چاپ کن. در تابع main ابتدا فایل باز شده و سپس تابع تحلیلگر لغوی فراخوانی شده و سپس فایل بسته می شود. در Lex هم مانند Yacc یک سری متغیرها از قبل تعریف شده اند که در جدول زیر می توانید آنها را مشاهده کنید.

Name	Function
int yylex(void)	تابع تحلیلگر لغوی که Token بر میگرداند
char *yytext	اشاره گر به متنی که به عنوان Token تشخیص داده شده است
yyleng	طول کلمه ای که تشخیص داده شده است
yylval	مقدار یا ارزش مرتبط با Token
int yywrap(void)	تابعی است که در پایان کار تحلیل لغوی فراخوانی می شود و در صورت موفقیت 1 برمی گرداند و در صورت عدم موفقیت 0 بر می گرداند.
FILE *yyout	فایل خروجی
FILE *yyin	فایل ورودی
ECHO	الگوی تشخیص داده شده را عینا در خروجی می نویسد

اما اجازه دهید به عنوان مثالی پیچیده تر از تحلیل لغوی با Lex مثال ماشین حسابی را که در ابتدا طرح کرده بودیم، ببینیم.

```
%{
#include <stdio.h>
 }%
number   [0-9]+\.? | [0-9]*\.[0-9]+
%%
[ ]          {  /* skip blanks /*}
{number}               { sscanf( yytext ,"%lf", &yyval );
                       return NUMBER; }
\n |.                  {  return yytext[0]; }
%%
int yywrap(void)
{
   return 1;
}
int main(void)
{
```

```
  yylex();
  return 0;
}
```

تابع ()wrap تابعی است که در انتهای تحلیل لغوی فراخوانی می شود.

Error Recovery در Lex و Yacc:

در تجزیه گر نحوی می توان Error recovery را توسط گسترشی به نام error انجام داد. ابتدا به ساکن، کاربرتصمیم می گیرد که کدام ترم های میانی عمده باید Error recovery داشته باشند. سپس کاربر به هر یک از این قوانین زبان، یک گسترش به فرم A → **error** s اضافه می کند.که در اینجا A ترم های میانی مهم و s رشته ای از سمبل های گرامر، و شاید رشته تهی باشد. کلمه error در Yacc یک کلمه رزرو شده است.به این ترتیب، Yacc با Error ها به صورت گسترشی معمولی از قوانین رفتار خواهد کرد. هنگامی که پارسر با error مواجه می شود،آنقدر از stack تجزیه، pop می کند تا به یک state برسد که مجموعه item های آن state گسترشی به فرم A → **error** s داشته باشد. در آن اگر s لاندا نباشد، (یعنی رشته تهی) Yacc آنقدر از ورودی می خواند تا به عضوی از s برسد. در اینجا این گسترش کاهش داده می شود و semantic action مربوط به این error که می تواند یک روتین error recovery تعریف شده توسط کاربر باشد، اجرا خواهد شد. اگر s تهی باشد، کاهش به A بلافاصله انجام می شود و عمل semantic مربوط به آن اجرا می شود. اگر s کاملا از ترمهای پایانی تشکیل شده باشد، پارسر آنقدر آنها را shift می دهد که بتواند کاهش دهد.

به عنوان مثال فرض کنید یک گسترش error به فرم زیر داریم:

stmt → **error** ;

پارسر با برخورد به error می داند که باید آنقدر از پشته pop می کند تا با State ای مواجه شود که یک Shift action روی توکن error داشته باشد. State 0 چنین حالتی است زیرا آیتم های آن به شکل زیر است:

Lines → **error** '\n'

پارسر توکن error را به روی پشته شیفت می دهد و از ورودی آنقدر می خواند که به '\n' برخورد کند.در این نقطه پارسر '\n' را شیفت می دهد و error '\n' را کاهش می دهد. و روتین بازگشت از خطا را اجرا می کند.